Seminarul 4

În acest seminar vom învăța despre excepții, parametrii default, cunvântul cheie const și polimorfism.

1 Excepții

Excepțiile oferă posibilitatea de a trata situații "speciale" la momentul rulării (erori la runtime) prin transferul execuției către o zonă de cod care va gestiona eroarea. Pentru a putea gestiona o execuție trebuie ca setul de instrucțiuni care ar putea genera excepția să fie inclus într-un bloc `try-catch`. Exemplu:

```
#include <iostream>
using namespace std;
int main () {
    try {
        int i;
        cin >> i;
        if (i % 2) {
            throw i;                // arunca i ca exceptie daca este impar
        }
        cout << i << " este par";
    } catch (int x) {               // gestioneaza exceptia
        cout << x << " este impar";
    }
    cout << endl;
    return 0;
}
```

Putem avea mai multe zone `catch` înlănțuite, fiecare oferind gestiunea unui tip diferit de excepție. Dacă nu stim ce tip de excepție se aruncă putem folosi `...` pentru a oferi o gestiune generică.

```
try {

}
catch (int i) {}        // gestioneaza intregi
catch (string s) {}     // gestioneaza stringuri
...
catch (...) {}          // gestioneaza tot ce nu e tratat mai sus
```

În C++ avem definită clasa abstractă `exception` (prezentă în headerul cu același nume) utilizată în gestiunea excepțiilor. Aceasta expune metoda `what` care întoarce `const char*` (o descriere a excepției). Este recomandat ca atunci când în code trebuie definite situații excepționale se recomandă aruncarea de excepții care moștenesc clasa `exception` (definite de utilizator sau nu). Tipuri de excepții deja definite:

- `bad_alloc`
- `bad_cast`
- `bad_exception`
- `bad_function_call`
- `bad_typeid`
- `bad_weak_ptr`
- `ios_base::failure`
- `logic_error`
- `runtime_error`
- `domain_error`
- `future_error`
- `invalid_argument`
- `length_error`
- `out_of_range`
- `overflow_error`
- `range_error`
- `system_error`
- `underflow_error`
- `bad_array_new_length`

2 Proprietăți, metode și obiecte const

Putem declara o variabilă ca fiind constantă (a cărei valoare nu se modifică) adăugând cuvântul cheie `const` înainte de a specifica tipul de date. Când declarăm o variabilă `const` compilatorul va încerca să nu aloce spațiu în memorie pentru acea variabilă.

```
int main () {
    const int i = 3;
    // i = 4;      eroare de compilare
    int j = i; // constanta se comporta ca orice variabila de tip intreg
    return 0;
}
```

Cuvântul cheie const poate fi amestecat cu pointeri și referințe. Putem declara pointer către constante, pointeri constanți dar și referințe către constante:

```
int main () {
    int i = 6;
    int b = 3;
    const int *p = &i;    // pointer catre o constanta de tip intreg
    // *p = 2;               eroare
    p = &b;               // functioneaza
    int* const cp = &i;   // pointer constant
    *cp = 6;              // functioneaza
    // cp = &b;              eroare
    const int &r1 = i;    // referinta catre o constanta
    return 0;
}
```

Într-o clasă putem avea proprietăți și metode `const` . Proprietățile constante pot fi inițializate cu parametri dați către constructor doar în lista de inițializare. Metodele `const` sunt metode care nu pot modifica starea obiectului, i.e. nu pot altera nici una dintre propietățile obiectului. O metodă poate fi declarată const prin adăugarea cuvântului cheie `const` după lista de parametrii. Exemplu:

```
class A {
    const int i;
    float f;
public:
    A (int a = 0, float b = 3.0) : i(a) {
        f = b;
        // i = (int) f - a; eroare de compilare
    }

    void foo () const { // metoda const
        int j = i + 22;
        // f = 33;          eroare de compilare
    }

    void bar () {
        f = 25;         // metoda nu e constanta, putem modifica
    }
};
```

Odată definită o clasă, putem declara obiecte constante de tipul clasei repective. Cu un obiect constant nu putem apela decât metode `const` .

```
int main () {
    const A a;
    // a.bar();   eroare de compilare
    a.foo();   // functioneaza
}
```

3 Parametrii default

In C++ putem atribui valori default pentru parametrii unei funcții. La momentul apelări dacă o valoare nu este funizată pentru un parametru, valoare default este folosită.
Exemplu

```
#include <iostream>

int sum (int a, int b = 0, c = 0) {
    return a + b;
```

```
}

int main () {
    // se apeleaza sum cu a = 4, b = 87 si c = 34
    std::cout << sum(4, 87, 34);
    // se apeleaza sum cu a = 10, b = 43 si c = 0 - valoare default
    std::cout << sum(10, 43);
    // se apeleaza sum cu a = 5, b = 0 si c = 0 - b si c valori default
    std::cout << sum(5);

    return 0;
}
```

Observții:

- parametrii default se poziționeaza mereu la capătul listei de parametri
- De asemenea, nu putem alterna între parametri default și parameteri non-default (această signatură este greșită: `int foo (int a = 1, int b);`)
- Nu putem alege care dintre parametrii funcției noastre să primească valori default și care nu (în cazul funcției `sum` dacă apelul se face cu doi parametrii, atunci parametrul `c` este cel care primește valoarea default).

4 Polimorfism

În C++ avem polimorfism pe funcții și metode. Prin asta înțelegem că putem să refolosim nume de funcții și metode pentru a declara noi funcții și metode, atât timp cât nu avem 2 definiții care au liste identice de parametrii (i.e. pentru a supraîncărca/redeclara o functie trebuie să furnizăm o nouă listă de parametrii). Exemplu:

```
// Polimorfism functii
#include <iostream>
using namespace std;

int multiply(int a, int b) {
    return a * b;
}

string multiply (int a, string b) {
    string result = "";
    for (int i = 0; i < a; i++) {
        result += b;
    }
    return result;
}

// o schimbare doar tipului de parameteri
// va rezulta int-o eroare de compilare
// float multiply (int a, int b) {
//     return float(a)*b;
// }

int main () {
    int x = multiply(3, 5);
    string s = multiply(4, "A");

    cout << x << " " << s << endl;  // 15 AAAA
    return 0;
}
```

```
// Polimorfism metode
#include <iostream>
using namespace std;

class C {
    int a;
    public:
    C(int);
    int multiply (int);
    string multiply (string);
    // float multiply (int);  - eroare de compilare
};

C::C(int x) : a(x) { }
```

```
15
16 int C::multiply (int b) {
17     return a * b;
18 }
19
20 string C::multiply (string s) {
21     string result = "";
22     for (int i = 0; i < a; i++) {
23         result += s;
24     }
25     return result;
26 }
27
28 int main () {
29     C c (3);
30     cout << c.multiply(5) << " " << c.multiply("A");    // 15 AAA
31     return 0;
32 }
```

5 Supraîncărcarea operatorilor

Datorită polimorfismului putem supraîncărca operatori pentru clase definite de utilizator. Supraîncărcarea nu presupune că putem schimba numărul de parametrii (aritatea) și nici precedența operatorului. Toți operatorii cunoscuți pot fi supraîncărcați cu excepția:

- `.` ;
- `::` ;
- `?:` ;
- `sizeof` ;

Când supraîncărcăm majoritatea operatorilor avem două opțiuni: 1. supraîncărcare ca metodă (putem vedea operatorul ca pe o metodă unde operandul din stânga este cel cu care se apelează metoda, iar operandul din dreapta este parametrul metodei) sau 2. supraîncărcare ca funcție prieten. Dacă, la supraîncarcare, primul operand nu are tipul clasei pentru care supraîncărcăm, atunci suntem obligați să supraîncărcăm ca funcție prieten. Dacă nu supraîncărcăm nici un operator, primim de la compilator operatorul `=` .

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4  class Point {
5      int x, y;
6      public:
7      Point (const int& a = 0, const int& b = 0) : x(a), y(b) { }
8      // supraincarcarea operatorului - pentru a
9      // determina distanta dintre doua puncte
10     double operator-(const Point&) const;
11     // supraincarcarea operatorului * pentru produsul
12     // scalar a doua puncte/doi vectori
13     int operator*(const Point&) const;
14     // supraincarcarea operatorului + pentru
15     // tranlatarea unui punct
16     friend Point operator* (const int&, const Point&);
17     // supraincarcarea operatorului << pentru afisarea unui punct
18     friend ostream& operator<<(ostream&, const Point&);
19     // supraincarcarea operatorului >> pentru citirea unui punct
20     friend istream& operator>>(istream&, Point&);
21 };
22
23 double Point::operator- (const Point& p) const {
24     double dx = x - p.x, dy = y - p.y;
25     double px = dx*dx, py = dy*dy;
26     return sqrt(px+py);
27 }
28
29 int Point::operator* (const Point& p ) const {
30     return x * p.x + y *p.y;
31 };
32
33 Point operator* (const int& x, const Point& p) {
```

```
34      Point o;
35      o.x = x * p.x;
36      o.y = x * p.y;
37      return o;
38  }
39
40  ostream& operator<<(ostream& out, const Point& p) {
41      out << "(" << p.x << "," << p.y << ")";
42      return out;
43  }
44
45
46  istream& operator>>(istream& in, Point& p) {
47      in >> p.x >> p.y;
48      return in;
49  }
50
51  int main () {
52      Point m(1, 6), n(12, 5);
53      cout << "Distanta dintre " << m << " si " << n << " este " << m - n << endl;
54      // Distanta dintre (1,6) si (12,5) este 11.0454
55      cout << "Translatam " << m << " cu 3 " << 3 * m << endl;
56      // Translatam (1,6) cu 3 (3,18)
57      cout << "Produsul scalar " << m * n << endl;
58      // Produsul scalar 42
59      return 0;
60  }
```